Dario Della Monica

Chapter 15: Query Processing

These slides are a modified version of the slides provided with the book:

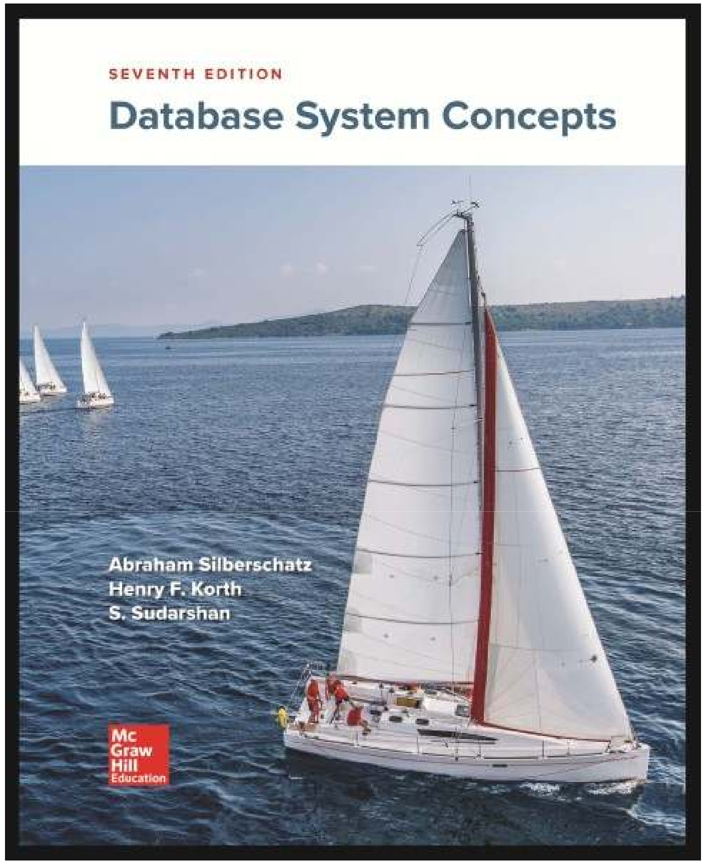
Database System Concepts, 6th Ed.

(however, chapter numeration refers to 7th Ed.)

The original version of the slides is available at: https://www.db-book.com/

Chapter 15: Query Processing

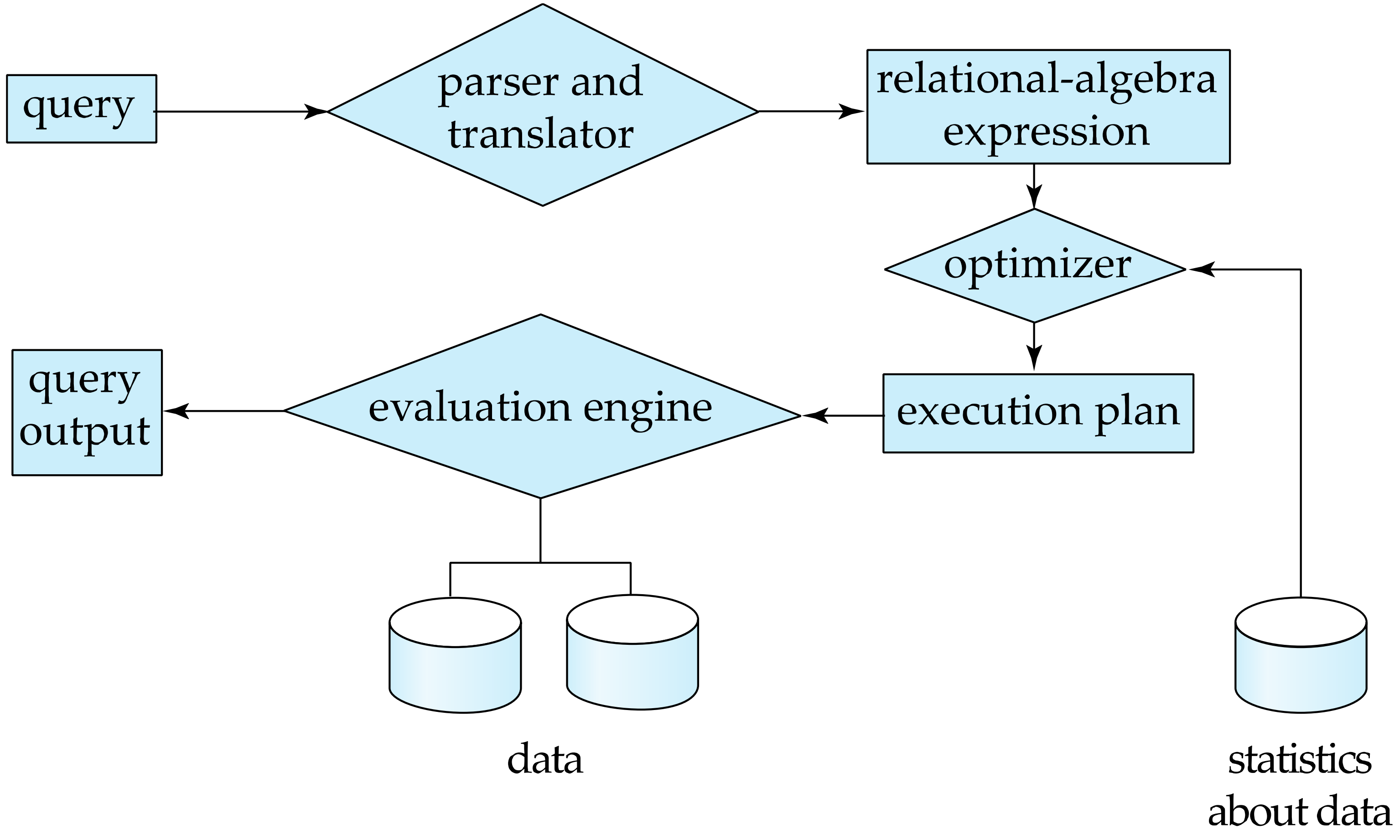
- Overview of query processing and query optimization
- How to measure query costs
  - Establishing a cost model
- Algorithms for relational algebra operations and their cost (estimates)
  - Selection
  - Sorting
  - Join
- Evaluation of Expressions (How to combine algorithms for individual operations in order to evaluate a complex expression)
  - Materialization
  - Pipelining



Silberschatz, Korth, Sudarshan, *Database System Concepts*, 7° edition, 2011

Basic Steps in Query Processing

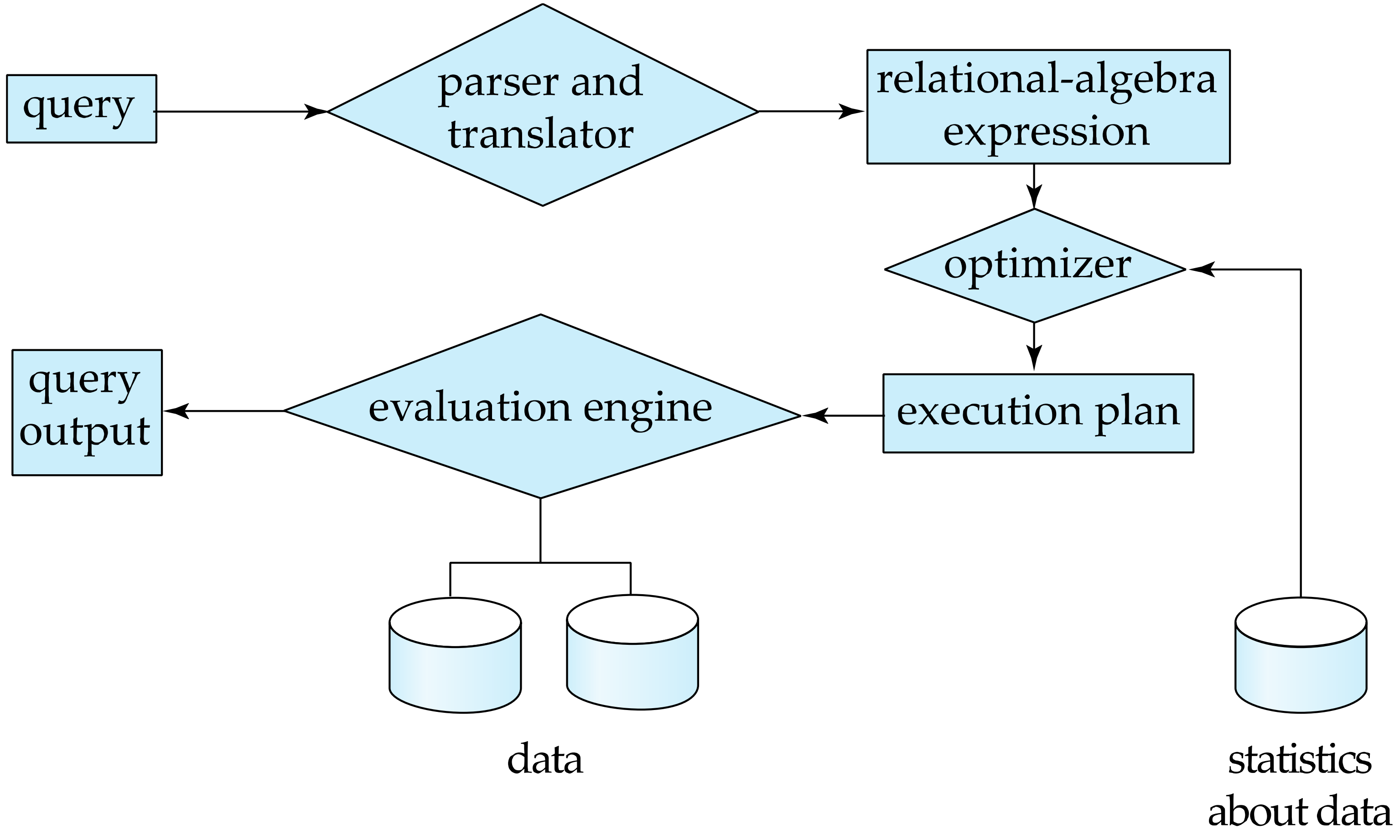
1. Parsing and translation
2. Optimization
3. Evaluation



Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation

We mainly focus on the optimization phase



Basic Steps in Query Processing (cont.)

- Parser and translator
  - Translate the (SQL) query into relational algebra
  - Parser checks syntax (e.g., correct relation and operator names)

- Evaluation engine
  - The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query

- Optimizer (in a nutshell – more details in the next slides)
  - Chooses the "most efficient" implementation to execute the query
    - Produces equivalent relational algebra expressions
    - Annotates them with instructions (algorithms): query execution plan (QEP)
    - Estimates the cost of "all" equivalent QEP, according to a suitable cost model
    - Choose the "best" QEP *(remember that there is a couple of "lies" in this sentence)*

Basic Steps: Optimization

- Input of optimization: a query in the form of an algebra expression
- Output of optimization: the "best" annotated relational algebra expression specifying detailed evaluation strategy (**query evaluation plan** **or** **query execution plan** – **QEP**) answering the input query
- 1st level of optimization: *order of operations* – an SQL query has many equivalent relational algebra expressions
  - Consider the SQL query

    ```
    SELECT salary
    FROM instructor
    WHERE salary < 75000
    ```
  - $\sigma_{salary<75000}(\Pi_{salary}(\textit{instructor}))$ and $\Pi_{salary}(\sigma_{salary<75000}(\textit{instructor}))$ are equivalent and **they both correspond to the above query**
- 2nd level of optimization: *implementation of operations* – a relational algebra operation can be evaluated via several different algorithms
  - e.g., block nested-loop join VS. merge-join; file scan VS. index scan

Basic Steps: Optimization (Cont.)

- Different query evaluation plans have different costs
  - Users are released from the burden of efficiency: they are not expected to say how to get what they want by specifying least-cost plans
- **Query Optimization:** amongst "all" equivalent QEP choose the one with "lowest" cost
  - Cost is estimated using meta-information or statistical information stored in or computed using the database **catalog**
    - # of tuples in relations, tuple sizes, # of distinct values for a given attribute, etc.
    - selectivity rate of a condition *A*=*x* where x is resolved at runtime
- This was an overview of the query processing phase. What's next?
  - Evaluation of "all" QEP – creation of the search space (Chapter 15[*])
    - How to measure query costs (establish a cost model)
    - Algorithms for relational algebra operations and their cost (estimates)
    - How to combine algorithms for individual operations in order to evaluate a complex expression (QEP)
  - Choosing the "best" QEP – efficiently explore the search space (Chapter 16[*])
    - How to optimize queries, that is, how to find a QEP with "lowest" estimated cost

[*] **Silberschatz, Korth, and Sudarshan, *Database System Concepts*, 7° ed.**

How to measure query costs (cost model)

Measures of Query Cost

Response time (wall-clock time needed to execute a plan) depends on several factors

- system configuration
  - amount of (dedicated) RAM (aka, memory, main memory)
  - type of hardware and/or system architecture
- runtime conditions
  - amount of free buffer at the time the plan is executed
  - content of the buffer at the time the plan is executed
  - parameters, embedded in queries, which are resolved at runtime only

```
SELECT salary
FROM instructor
WHERE salary < $a
```

where $a is a variable provided by the application (user)

Thus, an empirical analysis cannot be effective; a theoretical analysis is needed, based on a suitable cost model

1. every theoretical analysis must be recast with actual performance ofparameters used by the concrete system (hardware) to which the analysis is going to be applied
2. different optimizers may make different assumptions (different cost model)
3. cost models (like ours) focus on resource consumption rather than response time (optimizers minimize resource consumption rather than response time)

Measures of Query Cost (Cont.)

- Query cost (total elapsed time for answering a query) can be measured in terms of different resources
  - *disk access* (I/O operation on disk)
  - *CPU usage*
  - (*network communication* for distributed DBMS – later in this course)
- Typically **disk access** is the predominant cost, and is also relatively easy to estimate. Measured by taking into account
  - **Number of seeks** (number of random I/O accesses)
  - **Number of blocks read**
  - **Number of blocks written**
    - It is generally assumed cost for writing to be twice as the cost for reading (data is read back after being written to ensure the write was successful)

Measures of Query Cost (Cont.)

- Query cost (total elapsed time for answering a query) can be measured in terms of different resources
  - *disk access* (I/O operation on disk)
  - *CPU usage*
  - (*network communication* for distributed DBMS – later in this course)
- Typically **disk access** is the predominant cost, and is also relatively easy to estimate. Measured by taking into account
  - **Number of seeks** (number of random I/O accesses)
  - **Number of blocks read**
  - **Number of blocks written**
    - It is generally assumed cost for writing to be twice as the cost for reading (data is read back after being written to ensure the write was successful)

VERY IMPORTANT!!!
- "**disk**" refers to **permanent drive** for file storage, **hard-disk**, **secondary memory**, **permanent memory**
- "**memory**" refers to **volatile drive** for data storage, **RAM**, **main memory**, **buffer**
These are all used as synonims

Measures of Query Cost (Cont.)

- Query cost (total elapsed time for answering a query) can be measured in terms of different resources
  - *disk access* (I/O operation on disk)
  - *CPU usage*
  - (*network communication* for distributed DBMS – later in this course)
- Typically **disk access** is the predominant cost, and is also relatively easy to estimate. Measured by taking into account
  - **Number of seeks** (number of random I/O accesses)
  - **Number of blocks read**
  - **Number of blocks written**
    - It is generally assumed cost for writing to be twice as the cost for reading (data is read back after being written to ensure the write was successful)

VERY IMPORTANT!!!
- “**disk**” refers to **permanent drive** for file storage, **hard-disk**, **secondary memory**, **permanent memory**
- “**memory**” refers to **volatile drive** for data storage, **RAM**, **main memory**, **buffer**
These are all used as synonims

This is a **so far** accepted choice for measuring query costs (**cost model**).
New technologies: faster hard-disks (solid-state drives – SSD) and cheaper (thus bigger) RAM might direct towards different cost models (e.g., based also on CPU usage or RAM I/O operations)

Measures of Query Cost (Cont.)

- We ignore difference between writing and reading: we just consider
  - t_S – time for one **seek**
  - t_T – time to **transfer** one block
  - Example: cost for ***b*** block transfers plus ***S*** seeks

$$b * t_T + S * t_S$$

  - Values of t_T and t_S must be calibrated for the specific disk system
  - Typical values (2018): t_S = 4 ms, t_T = 0.1 ms
  - Some DBMS performs, during installation, seeks and block transfers to estimate average values
- We ignore CPU costs for simplicity
  - Real systems usually do take CPU cost into account
- We do not include cost to writing output to disk in our cost formulae

Algorithms for evaluating relational algebra operations

These slides are a modified version of the slides provided with the book:

Database System Concepts, 6th Ed.

(however, chapter numeration refers to 7th Ed.)

The original version of the slides is available at: https://www.db-book.com/

Physical organization of records

- At the physical level, relations are stored (on permanent disks) as files (managed and organized by the filesystem)
- We assume files are organized according to **sequential file organization**
  - i.e., a file is stored in contiguous blocks, with records ordered according to some attribute(s) – not necessarily ordered by primary key
- Other file organization techniques exist (e.g., B^+-tree file organization), leading to different formulas for cost estimate

Selection Operation

- **File scan** (relation scan without indices)

  PROs: can be applied to any file, regardless of its ordering, availability of indices, nature of selection operation, etc.

  CONs: it is slow

- Algorithm **A1** (**linear search**). Retrieve and scan each file block and test all records to see whether they satisfy the selection condition
  - b_r denotes number of blocks containing records from relation r
  - *Cost estimate???* (selection on a generic, non-key attribute)

Selection Operation

- **File scan** (relation scan without indices)

  PROs: can be applied to any file, regardless of its ordering, availability of indices, nature of selection operation, etc.

  CONs: it is slow

- Algorithm **A1** (**linear search**). Retrieve and scan each file block and test all records to see whether they satisfy the selection condition
  - b_r denotes number of blocks containing records from relation r
  - *Cost estimate???* (selection on a generic, non-key attribute)
    - cost = b_r block transfers + 1 seek = $t_S + b_r * t_T$

We assume blocks are stored contiguously so 1 seek operation is enough (disk head does not need to move to seek next block)

Selection Operation

- **File scan** (relation scan without indices)

  PROs: can be applied to any file, regardless of its ordering, availability of indices, nature of selection operation, etc.

  CONs: it is slow

- Algorithm **A1** (**linear search**). Retrieve and scan each file block and test all records to see whether they satisfy the selection condition
  - b_r denotes number of blocks containing records from relation r
  - *Cost estimate???* (selection on a generic, non-key attribute)
    - cost = b_r block transfers + 1 seek = $t_S + b_r * t_T$

We assume blocks are stored contiguously so 1 seek operation is enough (disk head does not need to move to seek next block)

  - Selection on a key attribute. *Cost estimate???*

Selection Operation

- **File scan** (relation scan without indices)

  PROs: can be applied to any file, regardless of its ordering, availability of indices, nature of selection operation, etc.

  CONs: it is slow

- Algorithm **A1** (**linear search**). Retrieve and scan each file block and test all records to see whether they satisfy the selection condition
  - b_r denotes number of blocks containing records from relation r
  - *Cost estimate???* (selection on a generic, non-key attribute)
    - cost = b_r block transfers + 1 seek = $t_S + b_r * t_T$

We assume blocks are stored contiguously so 1 seek operation is enough (disk head does not need to move to seek next block)

  - Selection on a key attribute. *Cost estimate???*
    - stop on finding record
    - cost = $(b_r/2)$ block transfers + 1 seek = $t_S + (b_r / 2) * t_T$

Selections Using Indices

- **Index scan** (relation scan using an index)
  - selection condition must be on search-key of index
  - h_i : height of the B^+-tree (# of accesses to traverse the index before accessing the data)

Selections Using Indices

- **Index scan** (relation scan using an index)
  - selection condition must be on search-key of index
  - ***h_i* :** height of the B^+-tree (# of accesses to traverse the index before accessing the data)
- **A2** (**primary index, equality on key**). Retrieve a single record that satisfies the corresponding equality condition. *Cost?*

Selections Using Indices

- **Index scan** (relation scan using an index)
  - selection condition must be on search-key of index
  - ***h_i* :** height of the B$^+$-tree (# of accesses to traverse the index before accessing the data)
- **A2** (**primary index, equality on key**). Retrieve a single record that satisfies the corresponding equality condition. *Cost?*
  - cost = $(h_i + 1) * (t_T + t_S)$

Selections Using Indices

- **Index scan** (relation scan using an index)
  - selection condition must be on search-key of index
  - ***h_i* :** height of the B$^+$-tree (# of accesses to traverse the index before accessing the data)
- **A2** (**primary index, equality on key**). Retrieve a single record that satisfies the corresponding equality condition. *Cost?*
  - cost = $(h_i + 1) * (t_T + t_S)$
- **A3** (**primary index, equality on nonkey**). Retrieve multiple records. *Cost?*

Selections Using Indices

- **Index scan** (relation scan using an index)
  - selection condition must be on search-key of index
  - h_i *:* height of the B$^+$-tree (# of accesses to traverse the index before accessing the data)
- **A2** (**primary index, equality on key**). Retrieve a single record that satisfies the corresponding equality condition. *Cost?*
  - cost = $(h_i + 1) * (t_T + t_S)$
- **A3** (**primary index, equality on nonkey**). Retrieve multiple records. *Cost?*
  - Let *b* = number of blocks containing matching records
    N.B.: ***b* is different from b_r (number of records in *r*) from previous slide**
  - Records will be on consecutive blocks

Selections Using Indices

- **Index scan** (relation scan using an index)
  - selection condition must be on search-key of index
  - h_i : height of the B$^+$-tree (# of accesses to traverse the index before accessing the data)
- **A2** (**primary index, equality on key**). Retrieve a single record that satisfies the corresponding equality condition. *Cost?*
  - cost = $(h_i + 1) * (t_T + t_S)$
- **A3** (**primary index, equality on nonkey**). Retrieve multiple records. *Cost?*
  - Let *b* = number of blocks containing matching records
    N.B.: *b* is different from b_r (number of records in *r*) from previous slide
  - Records will be on consecutive blocks
  - cost = $h_i * (t_T + t_S) + t_S + t_T * b$

Selections Using Indices

- **Index scan** (relation scan using an index)
  - selection condition must be on search-key of index
  - ***h_i :*** height of the B$^+$-tree (# of accesses to traverse the index before accessing the data)
- **A2** (**primary index, equality on key**). Retrieve a single record that satisfies the corresponding equality condition. *Cost?*
  - cost = $(h_i + 1) * (t_T + t_S)$
- **A3** (**primary index, equality on nonkey**). Retrieve multiple records. *Cost?*
  - Let *b* = number of blocks containing matching records
    N.B.: *b* is different from b_r (number of records in *r*) from previous slide
  - Records will be on consecutive blocks
  - cost = $h_i * (t_T + t_S) + t_S + t_T * \mathbf{b}$

There is a mistake in the **6th ed.** of the book* (Fig. 12.3): the "t_S" summand is omitted

* Silberschatz, Korth, and Sudarshan, *Database System Concepts*, 6° ed.

Selections Using Indices

- **A4** (**secondary index, equality on key**). *Cost?*

Selections Using Indices

- **A4** (**secondary index, equality on key**). *Cost?*
  - Equal to **A2**

Selections Using Indices

- **A4** (**secondary index, equality on key**). *Cost?*
  - Equal to **A2**
    - cost = $(h_i + 1) * (t_T + t_S)$

Selections Using Indices

- **A4** (**secondary index, equality on key**). *Cost?*
  - Equal to **A2**
    - cost = $(h_i + 1) * (t_T + t_S)$

- **A4** (**secondary index, equality on nonkey**)
  - Retrieve multiple records. *Cost?*

Selections Using Indices

- **A4** (**secondary index, equality on key**). *Cost?*
  - Equal to **A2**
    - cost = $(h_i + 1) * (t_T + t_S)$

- **A4** (**secondary index, equality on nonkey**)
  - Retrieve multiple records. *Cost?*
    - in the worst case, for each of *n* matching records a new block access (both transfer and seek) is needed (even if 2 records are in the same block but pointed by leaves that are far apart in the index tree)

Selections Using Indices

- **A4** (**secondary index, equality on key**). *Cost?*
  - Equal to **A2**
    - cost = $(h_i + 1) * (t_T + t_S)$
- **A4** (**secondary index, equality on nonkey**)
  - Retrieve multiple records. *Cost?*
    - in the worst case, for each of *n* matching records a new block access (both transfer and seek) is needed (even if 2 records are in the same block but pointed by leaves that are far apart in the index tree)
    - Cost = $(h_i + n) * (t_T + t_S)$
      - Can be very expensive! Can be worse than file scan

Selections Involving Comparisons

- Can implement selections of the form $\sigma_{A \leq V}(r)$ or $\sigma_{A \geq V}(r)$ by using
  - a linear file scan,
  - or by using indices in the following ways:

Selections Involving Comparisons

- Can implement selections of the form $\sigma_{A \leq V}(r)$ or $\sigma_{A \geq V}(r)$ by using
  - a linear file scan,
  - or by using indices in the following ways:
- **A5** (**primary index, comparison**).
  - $\sigma_{A \geq V}(r)$
  - $\sigma_{A \leq V}(r)$

Selections Involving Comparisons

- Can implement selections of the form $\sigma_{A \leq V}(r)$ or $\sigma_{A \geq V}(r)$ by using
  - a linear file scan,
  - or by using indices in the following ways:
- **A5** (**primary index, comparison**).
  - $\sigma_{A \geq V}(r)$
    - use index to find first tuple $\geq$ v and scan relation sequentially from there
    - RECALL: *b* is the number of blocks containing matching records
  - $\sigma_{A \leq V}(r)$

Selections Involving Comparisons

- Can implement selections of the form $\sigma_{A \leq V}(r)$ or $\sigma_{A \geq V}(r)$ by using
  - a linear file scan,
  - or by using indices in the following ways:
- **A5** (**primary index, comparison**).
  - $\sigma_{A \geq V}(r)$
    - use index to find first tuple $\geq$ v and scan relation sequentially from there
    - RECALL: *b* is the number of blocks containing matching records
    - Equal to **A3**: *Cost* = $h_i * (t_T + t_S) + t_S + t_T * b$
  - $\sigma_{A \leq V}(r)$

Selections Involving Comparisons

- Can implement selections of the form $\sigma_{A \leq V}(r)$ or $\sigma_{A \geq V}(r)$ by using
  - a linear file scan,
  - or by using indices in the following ways:
- **A5** (**primary index, comparison**).
  - $\sigma_{A \geq V}(r)$
    - use index to find first tuple $\geq$ v and scan relation sequentially from there
    - RECALL: *b* is the number of blocks containing matching records
    - Equal to **A3**: *Cost* = $h_i * (t_T + t_S) + t_S + t_T * b$
  - $\sigma_{A \leq V}(r)$
    - just scan relation sequentially till first tuple > v; do not use the index

Selections Involving Comparisons

- Can implement selections of the form $\sigma_{A \leq V}(r)$ or $\sigma_{A \geq V}(r)$ by using
  - a linear file scan,
  - or by using indices in the following ways:
- **A5** (**primary index, comparison**).
  - $\sigma_{A \geq V}(r)$
    - use index to find first tuple $\geq$ v and scan relation sequentially from there
    - RECALL: *b* is the number of blocks containing matching records
    - Equal to **A3**: *Cost* = $h_i * (t_T + t_S) + t_S + t_T * b$
  - $\sigma_{A \leq V}(r)$
    - just scan relation sequentially till first tuple > v; do not use the index
    - Similar to **A1** (file scan, equality on key): *Cost* = $t_S + b * t_T$

Selections Involving Comparisons

- **A6** (**secondary index, comparison**). *Cost?*

Selections Involving Comparisons

- **A6** (**secondary index, comparison**). *Cost?*
  - $\sigma_{A \geq V}(r)$
  - $\sigma_{A \leq V}(r)$

Selections Involving Comparisons

- **A6** (**secondary index, comparison**). *Cost?*
  - $\sigma_{A \geq V}(r)$
    - use index to find first index entry ≥ *v* and scan index sequentially from there, to find pointers to records, and finally retrieve records that are pointed to
  - $\sigma_{A \leq V}(r)$

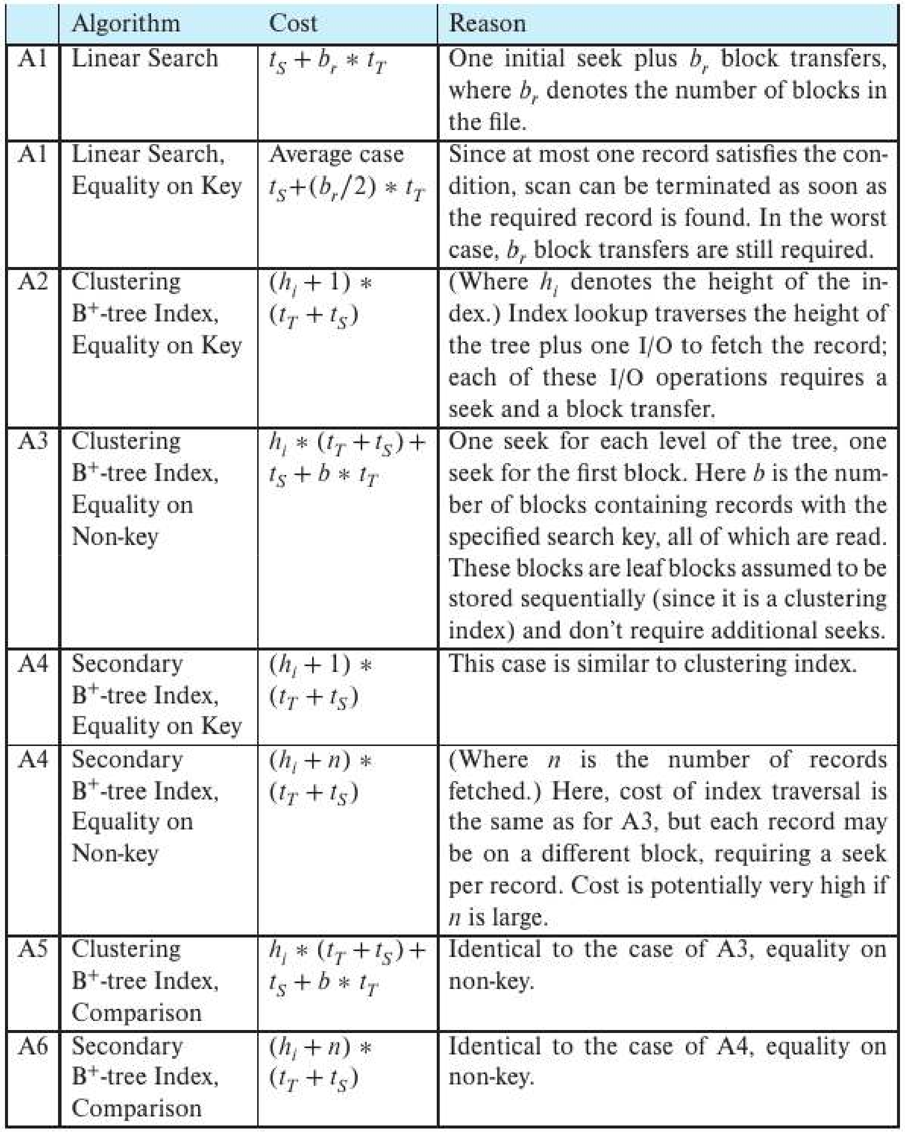
Selections Involving Comparisons

- **A6** (**secondary index, comparison**). *Cost?*
  - $\sigma_{A \geq V}(r)$
    - use index to find first index entry ≥ *v* and scan index sequentially from there, to find pointers to records, and finally retrieve records that are pointed to
    - requires an I/O for each record
    - Equal to **A4**, equality on nonkey: cost = $(h_i + n) * (t_T + t_S)$
    - Linear file scan may be cheaper
  - $\sigma_{A \leq V}(r)$

Selections Involving Comparisons

- **A6** (**secondary index, comparison**). *Cost?*
  - $\sigma_{A \geq V}(r)$
    - use index to find first index entry $\geq v$ and scan index sequentially from there, to find pointers to records, and finally retrieve records that are pointed to
    - requires an I/O for each record
    - Equal to **A4**, equality on nonkey: cost = $(h_i + n) * (t_T + t_S)$
    - Linear file scan may be cheaper
  - $\sigma_{A \leq V}(r)$
    - just scan leaf pages of index from the beginning up to the first entry > *v,* finding pointers to records, and finally retrieve records that are pointed to
    - equal to the above case

Summary of costs for selections



	Algorithm	Cost	Reason
A1	Linear Search	$t_S + b_r * t_T$	One initial seek plus b_r block transfers, where b_r denotes the number of blocks in the file.
A1	Linear Search, Equality on Key	Average case $t_S + (b_r/2) * t_T$	Since at most one record satisfies the condition, scan can be terminated as soon as the required record is found. In the worst case, b_r block transfers are still required.
A2	Clustering B$^+$-tree Index, Equality on Key	$(h_i + 1) * (t_T + t_S)$	(Where h_i denotes the height of the index.) Index lookup traverses the height of the tree plus one I/O to fetch the record; each of these I/O operations requires a seek and a block transfer.
A3	Clustering B$^+$-tree Index, Equality on Non-key	$h_i * (t_T + t_S) + t_S + b * t_T$	One seek for each level of the tree, one seek for the first block. Here b is the number of blocks containing records with the specified search key, all of which are read. These blocks are leaf blocks assumed to be stored sequentially (since it is a clustering index) and don't require additional seeks.
A4	Secondary B$^+$-tree Index, Equality on Key	$(h_i + 1) * (t_T + t_S)$	This case is similar to clustering index.
A4	Secondary B$^+$-tree Index, Equality on Non-key	$(h_i + n) * (t_T + t_S)$	(Where n is the number of records fetched.) Here, cost of index traversal is the same as for A3, but each record may be on a different block, requiring a seek per record. Cost is potentially very high if n is large.
A5	Clustering B$^+$-tree Index, Comparison	$h_i * (t_T + t_S) + t_S + b * t_T$	Identical to the case of A3, equality on non-key.
A6	Secondary B$^+$-tree Index, Comparison	$(h_i + n) * (t_T + t_S)$	Identical to the case of A4, equality on non-key.

Figure 15.3 Cost estimates for selection algorithms.

Complex Selections

conjunctions, disjunctions, and negation of simple conditions

- **A7** (**conjunctive selection using 1 index**)
  - θ_1 *AND* θ_2 *AND … AND* θ_n

Complex Selections

conjunctions, disjunctions, and negation of simple conditions

- **A7** (**conjunctive selection using 1 index**)
  - θ_1 *AND* θ_2 *AND … AND* θ_n
  - If there is at least 1 index useful for 1 simple condition θ_i, then
    - use the right algorithm among A2-A6 to retrieve tuple satisfying θ_i
    - and in the meantime check for the other simple conditions on records selected .
    - *Cost?*

Complex Selections

conjunctions, disjunctions, and negation of simple conditions

- **A7** (**conjunctive selection using 1 index**)
  - θ_1 *AND* θ_2 *AND … AND* θ_n
  - If there is at least 1 index useful for 1 simple condition θ_i, then
    - use the right algorithm among A2-A6 to retrieve tuple satisfying θ_i
    - and in the meantime check for the other simple conditions on records selected .
    - *Cost?*
  - Cost is given by the cost of chosen algorithm for the chosen condition
    - cost depend on the choice of the condition (and the choice of the algorithm)
    - example: $\sigma_{id=x \text{ AND } dept=y}$ (*teacher*)
    - primary index over *id* and secondary index over *dept*
    - *id* is primary key
    - it is convenient to choose *id*=*x* (with algorithm A2)

Complex Selections

conjunctions, disjunctions, and negation of simple conditions

- **A7** (**conjunctive selection using 1 index**)
  - θ_1 *AND* θ_2 *AND … AND* θ_n
  - If there is at least 1 index useful for 1 simple condition θ_i, then
    - use the right algorithm among A2-A6 to retrieve tuple satisfying θ_i
    - and in the meantime check for the other simple conditions on records selected .
    - *Cost?*
  - Cost is given by the cost of chosen algorithm for the chosen condition
    - cost depend on the choice of the condition (and the choice of the algorithm)
    - example: $\sigma_{id=x \text{ AND } dept=y}$ (*teacher*)
    - primary index over *id* and secondary index over *dept*
    - *id* is primary key
    - it is convenient to choose *id*=*x* (with algorithm A2)
- **A8** (**conjunctive selection using composite index**)
  - use a composite index over attributes involved in all or some of the simple conditions (it works smoothly with equivalences, what about other comparison operators?), if any
    - example: $\sigma_{name=x \text{ AND } dept=y}$ (*teacher*)
    - use composite index over pair *(name,dept)* with algorithm A4 (secondary index, equality on non-key)

Complex Selections (cont'd)

- **A9** (**conjunctive selection by intersection of identifiers**)
  - Recall that indices we consider have *pointers to records* (rather than *actual records*)
  - Scan indices but do not access records, just collect sets of pointers (one per index)
  - Compute the sorted intersection, and then access records **(at the end select only the tuples that match also the conditions for which there is no index)**. *Cost?*

Complex Selections (cont'd)

- **A9** (**conjunctive selection by intersection of identifiers**)
  - Recall that indices we consider have *pointers to records* (rather than *actual records*)
  - Scan indices but do not access records, just collect sets of pointers (one per index)
  - Compute the sorted intersection, and then access records **(at the end select only the tuples that match also the conditions for which there is no index)**. *Cost?*
  - Cost: cost of scanning indices plus cost of accessing blocks with matching records (some block can be accessed more than once if pointers are not sorted)
  - Advantages of sorting:
    - less seeks and transfers: no block is accessed twice (2 records in the same block are retrieved together)
    - also, some additional seek time is saved as blocks are transferred in sorted order (disk-arm is minimized)

Complex Selections (cont'd)

- **A9** (**conjunctive selection by intersection of identifiers**)
  - Recall that indices we consider have *pointers to records* (rather than *actual records*)
  - Scan indices but do not access records, just collect sets of pointers (one per index)
  - Compute the sorted intersection, and then access records **(at the end select only the tuples that match also the conditions for which there is no index)**. *Cost?*
  - Cost: cost of scanning indices plus cost of accessing blocks with matching records (some block can be accessed more than once if pointers are not sorted)
  - Advantages of sorting:
    - less seeks and transfers: no block is accessed twice (2 records in the same block are retrieved together)
    - also, some additional seek time is saved as blocks are transferred in sorted order (disk-arm is minimized)
- **A10** (**disjunctive selection by union of identifiers**)

Complex Selections (cont'd)

- **A9** (**conjunctive selection by intersection of identifiers**)
  - Recall that indices we consider have *pointers to records* (rather than *actual records*)
  - Scan indices but do not access records, just collect sets of pointers (one per index)
  - Compute the sorted intersection, and then access records **(at the end select only the tuples that match also the conditions for which there is no index)**. *Cost?*
  - Cost: cost of scanning indices plus cost of accessing blocks with matching records (some block can be accessed more than once if pointers are not sorted)
  - Advantages of sorting:
    - less seeks and transfers: no block is accessed twice (2 records in the same block are retrieved together)
    - also, some additional seek time is saved as blocks are transferred in sorted order (disk-arm is minimized)
- **A10** (**disjunctive selection by union of identifiers**)
  - If ALL conditions can be checked through some index, then similar to A9
    - Scan indices but do not access records, just collect sets of pointers (one per index)
    - Compute the union, and then access records (in sorted order)
    - Cost: cost of scanning all indices plus cost of accessing records

Complex Selections (cont'd)

- **A9** (**conjunctive selection by intersection of identifiers**)
  - Recall that indices we consider have *pointers to records* (rather than *actual records*)
  - Scan indices but do not access records, just collect sets of pointers (one per index)
  - Compute the sorted intersection, and then access records **(at the end select only the tuples that match also the conditions for which there is no index)**. *Cost?*
  - Cost: cost of scanning indices plus cost of accessing blocks with matching records (some block can be accessed more than once if pointers are not sorted)
  - Advantages of sorting:
    - less seeks and transfers: no block is accessed twice (2 records in the same block are retrieved together)
    - also, some additional seek time is saved as blocks are transferred in sorted order (disk-arm is minimized)
- **A10** (**disjunctive selection by union of identifiers**)
  - If ALL conditions can be checked through some index, then similar to A9
    - Scan indices but do not access records, just collect sets of pointers (one per index)
    - Compute the union, and then access records (in sorted order)
    - Cost: cost of scanning all indices plus cost of accessing records
  - If even only 1 condition has no associate index, then A1 (linear scan)

2 more things on selections

1. Negation of a simple condition

2 more things on selections

1. Negation of a simple condition
   - *NOT (Attr* < *v)* is equivalent to *Attr* >= *v* *NOT (Attr* > *v)* is equivalent to *Attr* <= *v*
   - *NOT (Attr* <= *v)* is equivalent to *Attr* > *v* *NOT (Attr* >= *v)* is equivalent to *Attr* < *v*

2 more things on selections

1. Negation of a simple condition
   - *NOT (Attr < v)* is equivalent to *Attr >= v* *NOT (Attr > v)* is equivalent to *Attr <= v*
   - *NOT (Attr <= v)* is equivalent to *Attr > v* *NOT (Attr >= v)* is equivalent to *Attr < v*
   - *NOT (Attr = v)* is equivalent *(Attr < v) OR (Attr > v)* – (probably a linear file scan is needed)

2 more things on selections

1. Negation of a simple condition
   - *NOT (Attr* < *v)* is equivalent to *Attr* >= *v* *NOT (Attr* > *v)* is equivalent to *Attr* <= *v*
   - *NOT (Attr* <= *v)* is equivalent to *Attr* > *v* *NOT (Attr* >= *v)* is equivalent to *Attr* < *v*
   - *NOT (Attr* = *v)* is equivalent *(Attr* < *v) OR (Attr* > *v)* – (probably a linear file scan is needed)

   Negation of a complex condition can be “pushed inside” the expressionn

2 more things on selections

1. Negation of a simple condition
   - *NOT (Attr* < *v)* is equivalent to *Attr* >= *v* *NOT (Attr* > *v)* is equivalent to *Attr* <= *v*
   - *NOT (Attr* <= *v)* is equivalent to *Attr* > *v* *NOT (Attr* >= *v)* is equivalent to *Attr* < *v*
   - *NOT (Attr* = *v)* is equivalent *(Attr* < *v) OR (Attr* > *v)* – (probably a linear file scan is needed)

   Negation of a complex condition can be "pushed inside" the expressionn
   - *NOT* (c_1 *AND* c_2) is equivalent to (*NOT* c_1) *OR* (*NOT* c_2)

2 more things on selections

1. Negation of a simple condition
   - *NOT (Attr < v)* is equivalent to *Attr >= v* *NOT (Attr > v)* is equivalent to *Attr <= v*
   - *NOT (Attr <= v)* is equivalent to *Attr > v* *NOT (Attr >= v)* is equivalent to *Attr < v*
   - *NOT (Attr = v)* is equivalent *(Attr < v) OR (Attr > v)* – (probably a linear file scan is needed)

   Negation of a complex condition can be "pushed inside" the expressionn
   - *NOT* $(c_1$ *AND* $c_2)$ is equivalent to (*NOT* c_1) *OR* (*NOT* c_2)

2. A4, A6, A9, A10 are very inefficient (possibly worse than linear scan) due to some blocks possibly accessed more than once
   - few records to be retrieved/no block is accessed more than once: better to use index scan,
     a lot of records to be retrieved/several blocks accessed more than once: better to use liner scan
   - Solution: collect and sort pointers before accessing records (see optimization for A9)
   - Improved solution, based on bitmap structure: bitmap is a vector of bits (as many as number of blocks used by the relation)
     - visit index without accessing records: in the bitmap set to 1 the bits corresponding to blocks to be accessed
     - linear scan guided by bitmap (sorted order access thanks to bitmap without actually performing a sorting)
     - hybrid solution (mix between linear scan and index access)
     - no block accessed more than once: slightly worse than index access,
       all blocks containing at least one matching condition: slightly worse than liner scan
     - thus, the cost is slightly worse than the optimal plan (linear or index scan): good compromise

Sorting

- Reasons for sorting
  - Explicitly requested by SQL query
    - SELECT …
      FROM …
      SORT BY …
  - Needed to efficient executions of join operations
- We may build an index on the relation, and then use the index to read the relation in sorted order. May lead to one disk block access for each tuple
- For relations that fit in memory, standard sorting techniques like quick-sort can be used. For relations that don't fit in memory, **external sort-merge** algorithm is a good choice

External Sort-Merge

Let M denote number of blocks that can fit in memory.

1. Create sorted **runs** (files containing sorted pieces of relation)

   Let i be 0 initially.

   Repeatedly do the following till the end of the relation:

   (a) Read M blocks of relation into memory

   (b) Sort the in-memory blocks

   (c) Write sorted data to run R_i

   (d) Increment i

   Let the final value of i be N *(number of runs)*

2. Merge the runs *(next slide)…..*

External Sort-Merge (Cont.)

2. Merge the runs (N-way merge). We assume (for now) that $N < M$.
    1. Use N blocks of memory to buffer input runs, and 1 block to buffer output. Read the first block of each run into its buffer page
    2. **repeat**
        1. Select the first record (in sort order) among the N blocks for the runs
        2. Write the record to the output buffer. If the output buffer is full write it to disk.
        3. Delete the record from its input buffer block.
           If the buffer block becomes empty **then**
             transfer the next block (if any) of the run into the buffer.

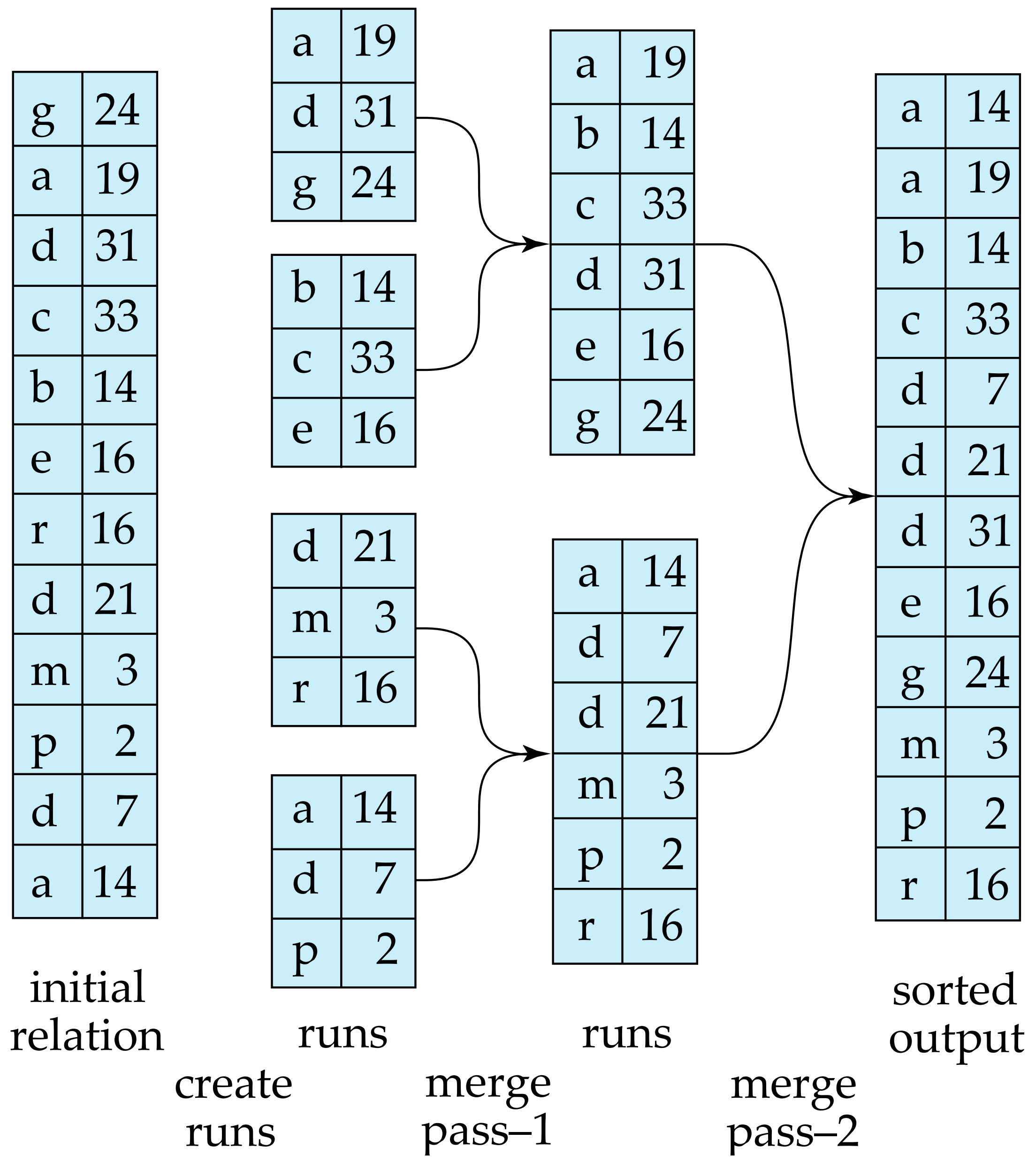
    until all blocks of all runs are processed

External Sort-Merge (Cont.)

- If $N \geq M$, several merge *passes* are required.
  - In each pass, contiguous groups of M - 1 runs are merged.
  - A pass reduces the number of runs by a factor of M -1 (and creates runs longer by the same factor)
    - E.g. If M=11, and there are 90 runs, one pass merge together groups of 10 runs into 9 new runs
      Thus, one pass reduces the number of runs to 9, each 10 times the size of the initial runs
  - Repeated passes are performed till all runs have been merged into one.

Example: External Sorting Using Sort-Merge

M = 3



External Sort-Merge: Cost Analysis

- Cost of block transfers:
  - Number of block transfers (read and write) for initial run creation: $2b_r$
  - Total number of merge passes required: $\lceil \log_{M-1}(b_r / M) \rceil$
  - Number of block transfers (read and write) in each pass: $2b_r$
  - For final pass, we don't count write cost: $- b_r$ (i.e, subtract b_r)
    - we ignore final write cost for all operations since the output of an operation may be sent to the parent operation without being written to disk or just shown to video
  - Thus, total number of block transfers for external sorting:

    $2b_r + 2\,b_r \lceil \log_{M-1} (b_r / M) \rceil - b_r =$

    $= b_r (2 \lceil \log_{M-1} (b_r / M) \rceil + 1)$

  - Seeks: next slide

External Sort-Merge: Cost Analysis (cont.)

- Cost of seeks
  - During run generation: one seek to read each run and one seek to write each run
    - $2\lceil b_r / M \rceil$
  - Total number of merge passes required: $\lceil \log_{M-1}(b_r / M) \rceil$
  - During each pass (merge phases)
    - 1 seek for reading each block and 1 seek for writing each block
      - $2\ b_r$ seeks for each merge pass
    - except the final one which does not require a write
      - $-\ b_r$ (i.e, subtract b_r)
  - Total number of seeks:

$$2\lceil b_r / M \rceil + 2b_r\lceil \log_{M-1}(b_r / M) \rceil - b_r \quad =$$

$$= \quad 2\lceil b_r / M \rceil + \ b_r\ (2\lceil \log_{M-1}(b_r / M) \rceil -1)$$

External Sort-Merge: Cost Analysis (cont.)

An improved version of the algorithm [*]

- Number of seeks can be reduced by using ***k*** many blocks (instead of 1) for each run during the run merge phase
  - Using 1 block per run leads to too many seeks
  - Instead, using *k* buffer blocks per run ➔ read/write *k* blocks with only 1 seek
    - Number of runs merged together: $\lfloor M/k \rfloor - 1$ runs (instead of $M - 1$)
      - Scaling factor is $\lfloor M / k \rfloor - 1$ instead of $M - 1$
    - Number of passes required: $\lceil \log_{\lfloor M / k \rfloor - 1}(b_r / M) \rceil$ instead of $\lceil \log_{M-1}(b_r / M) \rceil$
    - During the merge phase: $2 \lceil b_r / k \rceil$ seeks for each pass (instead of $2\ b_r$)
      - Except the final one (we assume final result is not written to disk)
- Thus total number of block transfers for external sorting:

  $$b_r (2 \lceil \log_{\lfloor M / k \rfloor -1} (b_r / M) \rceil + 1)$$

  Total number of seeks:

  $$2 \lceil b_r / M \rceil + \lceil b_r / k \rceil (2 \lceil \log_{\lfloor M / k \rfloor -1} (b_r / M) \rceil -1)$$

[*] In Silberschatz, Korth, and Sudarshan, Database System Concepts, 6° ed., the non-improved version of the algorithm is given only, but the cost analysis mixes elements from both versions of the algorithm

Join Operation

- Several different algorithms to implement joins
  - Nested-loop join
  - Block nested-loop join
  - Indexed nested-loop join
  - Merge-join
  - Hash-join
- Choice based on cost estimate
- Running example : *students* $\bowtie$ *takes*

  where

Number of *records* of	*student*:	5,000
Number of *blocks* of	*student*:	100
Number of *records* of	*takes*:	10,000
Number of *blocks* of	*takes*:	400

Nested-Loop Join

- To compute the theta join $r \bowtie_{\theta} s$

```
for each tuple $t_r$ in $r$ do
 for each tuple $t_s$ in $s$ do
     test pair $(t_r, t_s)$ to see if they satisfy the join condition $\theta$
     if it does, add $t_r \cdot t_s$ to the result
 end
end
```

- *r* is called the **outer relation** and *s* the **inner relation** of the join
- Requires no indices and can be used with any kind of join condition
- Expensive since it examines every pair of tuples in the two relations

Nested-Loop Join (Cont.)

- If the smaller relation fits entirely in memory, use that as the inner relation
  - $b_r + b_s$ block transfers and 2 seeks

  (same cost in the best case scenario, when both relations fit in memory)
- Worst case (there is enough memory forn only one block for each relation)

  # of block transfer:

Nested-Loop Join (Cont.)

- If the smaller relation fits entirely in memory, use that as the inner relation
  - $b_r + b_s$ block transfers and 2 seeks

  (same cost in the best case scenario, when both relations fit in memory)

- Worst case (there is enough memory forn only one block for each relation)

  # of block transfer:

  $n_r * b_s + b_r$ (*b_r* transfers to read relation *r* + $n_r * b_s$ transfers to read *s* for each tuple in *r*)

Nested-Loop Join (Cont.)

- If the smaller relation fits entirely in memory, use that as the inner relation
  - $b_r + b_s$ block transfers and 2 seeks

  (same cost in the best case scenario, when both relations fit in memory)

- Worst case (there is enough memory forn only one block for each relation)

  # of block transfer:

  $n_r * b_s + b_r$ (b_r transfers to read relation r + $n_r * b_s$ transfers to read s for each tuple in r)

  # of seeks:

Nested-Loop Join (Cont.)

- If the smaller relation fits entirely in memory, use that as the inner relation
  - $b_r + b_s$ block transfers and 2 seeks

  (same cost in the best case scenario, when both relations fit in memory)

- Worst case (there is enough memory forn only one block for each relation)

  # of block transfer:

  $n_r * b_s + b_r$ (b_r transfers to read relation r + $n_r * b_s$ transfers to read s for each tuple in r)

  # of seeks:

  $n_r + b_r$ (b_r seeks to read relation r + n_r seeks to read s for each tuple in r)

Nested-Loop Join (Cont.)

- If the smaller relation fits entirely in memory, use that as the inner relation
  - $b_r + b_s$ block transfers and 2 seeks

  (same cost in the best case scenario, when both relations fit in memory)

- Worst case (there is enough memory forn only one block for each relation)

  # of block transfer:

  $n_r * b_s + b_r$ (b_r transfers to read relation *r* + $n_r * b_s$ transfers to read *s* for each tuple in *r*)

  # of seeks:

  $n_r + b_r$ (b_r seeks to read relation *r* + n_r seeks to read *s* for each tuple in *r*)

- Running example (join between *students* and *takes* – assuming worst case memory availability)
  - with *student* as outer relation:
    - 5,000 ∗ 400 + 100 = 2,000,100 block transfers
    - 5,000 + 100 = 5,100 seeks
    - assuming $t_S = 40 * t_T$, we have a total of: **2,204,100 * t_T**

Nested-Loop Join (Cont.)

- If the smaller relation fits entirely in memory, use that as the inner relation
  - $b_r + b_s$ block transfers and 2 seeks

  (same cost in the best case scenario, when both relations fit in memory)

- Worst case (there is enough memory forn only one block for each relation)

  # of block transfer:

  $n_r * b_s + b_r$ (b_r transfers to read relation *r* + $n_r * b_s$ transfers to read *s* for each tuple in *r*)

  # of seeks:

  $n_r + b_r$ (b_r seeks to read relation *r* + n_r seeks to read *s* for each tuple in *r*)

- Running example (join between *students* and *takes* – assuming worst case memory availability)
  - with *student* as outer relation:
    - 5,000 ∗ 400 + 100 = 2,000,100 block transfers
    - 5,000 + 100 = 5,100 seeks
    - assuming $t_S = 40 * t_{T,}$ we have a total of: **2,204,100 * t_T**
  - with *takes* as the outer relation
    - 10,000 ∗ 100 + 400 = 1,000,400 block transfers
    - 10,000 + 400 = 10,400 seeks
    - assuming $t_S = 40 * t_{T,}$ we have a total of: **1,416,400 * t_T**

Nested-Loop Join (Cont.)

- If the smaller relation fits entirely in memory, use that as the inner relation
  - $b_r + b_s$ block transfers and 2 seeks

  (same cost in the best case scenario, when both relations fit in memory)

- Worst case (there is enough memory forn only one block for each relation)

  # of block transfer:

  $n_r * b_s + b_r$ (b_r transfers to read relation r + $n_r * b_s$ transfers to read s for each tuple in r)

  # of seeks:

  $n_r + b_r$ (b_r seeks to read relation r + n_r seeks to read s for each tuple in r)

- Running example (join between *students* and *takes* – assuming worst case memory availability)
  - with *student* as outer relation:
    - 5,000 ∗ 400 + 100 = 2,000,100 block transfers
    - 5,000 + 100 = 5,100 seeks
    - assuming $t_S = 40 * t_{T,}$ we have a total of: **2,204,100 * t_T**
  - with *takes* as the outer relation
    - 10,000 ∗ 100 + 400 = 1,000,400 block transfers
    - 10,000 + 400 = 10,400 seeks
    - assuming $t_S = 40 * t_{T,}$ we have a total of: **1,416,400 * t_T**
  - if smaller relation (*student*) fits entirely in memory, the cost estimate will be 500 block transfers and 2 seeks

Nested-Loop Join (Cont.)

- If the smaller relation fits entirely in memory, use that as the inner relation
  - $b_r + b_s$ block transfers and 2 seeks

  (same cost in the best case scenario, when both relations fit in memory)

- Worst case (there is enough memory forn only one block for each relation)

  # of block transfer:

  $n_r * b_s + b_r$ (b_r transfers to read relation *r* + $n_r * b_s$ transfers to read *s* for each tuple in *r*)

  # of seeks:

  $n_r + b_r$ (b_r seeks to read relation *r* + n_r seeks to read *s* for each tuple in *r*)

- Running example (join between *students* and *takes* – assuming worst case memory availability)
  - with *student* as outer relation:
    - 5,000 ∗ 400 + 100 = 2,000,100 block transfers
    - 5,000 + 100 = 5,100 seeks
    - assuming $t_S = 40 * t_T$, we have a total of: **2,204,100 * t_T**
  - with *takes* as the outer relation
    - 10,000 ∗ 100 + 400 = 1,000,400 block transfers
    - 10,000 + 400 = 10,400 seeks
    - assuming $t_S = 40 * t_T$, we have a total of: **1,416,400 * t_T**
  - if smaller relation (*student*) fits entirely in memory, the cost estimate will be 500 block transfers and 2 seeks

- Block nested-loops algorithm (next slide) is preferable

Block Nested-Loop Join

- Variant of nested-loop join in which every block of inner relation is paired with every block of outer relation.

```
for each block $B_r$ of $r$ do
    for each block $B_s$ of $s$ do
        for each tuple $t_r$ in $B_r$ do
            for each tuple $t_s$ in $B_s$ do
                check if $(t_r,t_s)$ satisfy the join condition
                if it does, add $t_r \cdot t_s$ to the result
            end
        end
    end
end
```

Block Nested-Loop Join (Cont.)

- Worst case estimate (memory holds one block for each relation):
  - Each block in the inner relation is read once for each block in the outer relation
  - # of block transfers:
  - # of seeks:

Block Nested-Loop Join (Cont.)

- Worst case estimate (memory holds one block for each relation):
  - Each block in the inner relation is read once for each block in the outer relation
  - # of block transfers:
    - $b_r * b_s + b_r$
  - # of seeks:

Block Nested-Loop Join (Cont.)

- Worst case estimate (memory holds one block for each relation):
  - Each block in the inner relation is read once for each block in the outer relation
  - # of block transfers:
    - $b_r * b_s + b_r$
  - # of seeks:
    - $2 * b_r$

Block Nested-Loop Join (Cont.)

- Worst case estimate (memory holds one block for each relation):
  - Each block in the inner relation is read once for each block in the outer relation
  - # of block transfers:
    - $b_r * b_s + b_r$
  - # of seeks:
    - $2 * b_r$
- (block) nested-loop improvements
  - If join attributes form a key for the inner relation
    - inner loop terminates when first match is found

Block Nested-Loop Join (Cont.)

- Worst case estimate (memory holds one block for each relation):
  - Each block in the inner relation is read once for each block in the outer relation
  - # of block transfers:
    - $b_r * b_s + b_r$
  - # of seeks:
    - $2 * b_r$
- (block) nested-loop improvements
  - If join attributes form a key for the inner relation
    - inner loop terminates when first match is found
  - If there is more space in memory

Block Nested-Loop Join (Cont.)

- Worst case estimate (memory holds one block for each relation):
  - Each block in the inner relation is read once for each block in the outer relation
  - # of block transfers:
    - $b_r * b_s + b_r$
  - # of seeks:
    - $2 * b_r$
- (block) nested-loop improvements
  - If join attributes form a key for the inner relation
    - inner loop terminates when first match is found
  - If there is more space in memory
    - read as many blocks as possible (say *k)* for the outer relation
    - # of block transfer: $\lceil b_r / k \rceil * b_s + b_r$
    - # of seeks: $2 * \lceil b_r / k \rceil$

Block Nested-Loop Join (Cont.)

- Worst case estimate (memory holds one block for each relation):
  - Each block in the inner relation is read once for each block in the outer relation
  - # of block transfers:
    - $b_r * b_s + b_r$
  - # of seeks:
    - $2 * b_r$
- (block) nested-loop improvements
  - If join attributes form a key for the inner relation
    - inner loop terminates when first match is found
  - If there is more space in memory
    - read as many blocks as possible (say *k)* for the outer relation
    - # of block transfer: $\lceil b_r / k \rceil * b_s + b_r$
    - # of seeks: $2 * \lceil b_r / k \rceil$
  - alternate forward and backward scan for inner relation
    - Use blocks already in buffer: save some block transfer

Indexed Nested-Loop Join

- If an index is available for one of the relations on the attribute of the join condition
  - then use such a relation as inner relation in a nested-loop join
- Instead of doing a linear scan as inner loop, do an index scan

```
for each tuple $t_r$ in $r$ do
        index scan over $s$ to find tuples $t_s$ satisfying the join condition with tuple $t_r$
end
```

(basically, for every tuple in *r*, do a selection on *s* using the index)

- It might be convenient to create an ad-hoc index for the join if it does not exist
- Cost (worst case: space in memory for only 1 block for each relation)
  - b_r seeks and block transfers to read *r*: $b_r * (t_T + t_S)$
  - for each record in *r*, index scan on *s*: $n_r * c$ (where *c* is the cost of index scan on *s*)
  - thus, total cost = $b_r * (t_T + t_S) + n_r * c$
  - NOTICE: if there are indexes for both relations, then it is often better to use relation with less records as outer relation

Indexed vs. Block Nested-Loop Join

- Block nested-loop join
  - $b_r * b_s + b_r$ block transfer and $2 * b_r$ seeks, that is, $(b_r * b_s + b_r) * t_T + 2 * b_r * t_S$
- Indexed nested-loop join
  - $b_r * (t_T + t_S) + n_r * c$
- Running example : *students* ⋈ *takes*
  - $n_{students} = 5{,}000$
    $b_{students} = 100$
    $n_{takes} = 10{,}000$
    $b_{takes} = 400$

    students is used as outer relation in block nested-loop join as it occupies less blocks
  - B^+-index as secondary index on *takes.student_id* with *fan* = *20*, and thus height *h* = *4*
  - cost of each index scan $c = (h + n) * (t_T + t_S)$ [A4: secondary index, eq. on non-key]
  - $n = n_{students} / n_{takes} = 2$ [average]
  - cost indexed nested-loop join (assuming $t_S = 40 * t_T$):
    $100 * (t_T + t_S) + 5{,}000 * ((4+2) * (t_T + t_S)) = 30{,}100 * (t_T + t_S) = 1{,}234{,}100 * t_T$
  - cost block nested-loop join (assuming $t_S = 40 * t_T$):
    $(100 * 400 + 100) * t_T + 2 * 100 * t_S = 40{,}100 * t_T + 200 * t_S = 48{,}100 + t_T$

How to combine algorithms for individual operations in order to evaluate a complex expression

Evaluation of Expressions

- So far: we have seen algorithms for individual operations
- Alternatives for evaluating an entire expression tree
  - **Materialization**: store (**materialize**) on disk results of evaluation of sub-expressions into temporary relations for subsequent use
    - Disadvantage: several disk writing and reading to store temporary relations
    - Always possible
  - **Pipelining**: pass on tuples to parent operations as they are generated by inner operations being executed
    - Advantage: less disk writing
    - Not always possible

Materialization

- **Materialized evaluation**: evaluate one operation at a time, starting at the lowest-level. Use intermediate results materialized into temporary relations to evaluate next-level operations

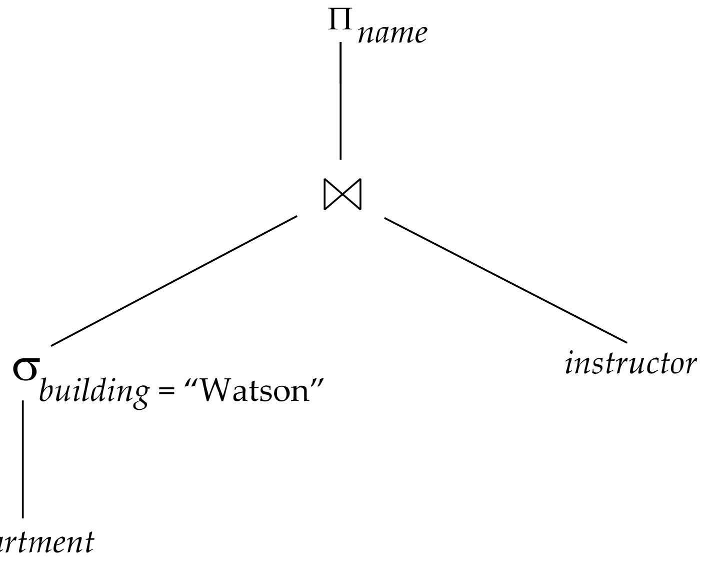
E.g., $\prod_{name}(\sigma_{building="\text{Watson}"}(department) \bowtie \text{instructor})$

1. compute and store on disk

   $\sigma_{building="\text{Watson}"}(department)$

2. then compute and store on disk its join with *instructor*

3. finally, compute the projection on *name*



Pipelining

- **Pipelined evaluation:** evaluate several operations simultaneously, passing (partial) results of one operation on to the next as they are generated (es., single records), without writing them on disk
- E.g., in previous expression tree, don't store result of

  $$\sigma_{building = \text{"Watson"}}(department)$$

  Instead, pass tuples directly to the join as they are found

  Similarly, don't store result of join, pass tuples directly to projection as they are generated
- Cheaper than materialization: no need to store a temporary relation to disk
- (partial) Results are output earlier, before waiting for complete query execution
- Parallelization of operations
- Pipelining may not always be possible
  - indexed nested-loop join cannot have its input inner relation pipelined as the whole relation with associated index must be available
  - some sorting algorithms cannot output tuples early, only after all input tuples have been examined (the curious case of the External Sort-Merge)

End of Chapter